10-1-2009

Interface Design For Programmable Cameras

Clifford Lindsay
Worcester Polytechnic Institute, clindsay@wpi.edu

Robert W. Linderman
Worcester Polytechnic Institute, gogo@wpi.edu

Emmanuel Agu
Worcester Polytechnic Institute, emmanuel@cs.wpi.edu

WPI-CS-TR-09-10 October 2009

Interface Design For Programmable Cameras

by

Clifford Lindsay
Robert W. Lindeman
Emmanuel Agu

WPI

Computer Science Technical Report Series

WORCESTER POLYTECHNIC INSTITUTE

Computer Science Department
100 Institute Road, Worcester, Massachusetts 01609-2280

Interface Design For Programmable Cameras

Clifford Lindsay *
Worcester Polytechnic Institute

Robert W. Lindeman†
Worcester Polytechnic Institute

Emmanuel Agu‡
Worcester Polytechnic Institute

Abstract

This paper outlines the process of designing a set of interfaces for a programmable digital camera back-end. The set of interfaces includes a traditional, advanced and development interface. The traditional interface is a replication of a typical camera interface which is present on today's commodity cameras. The advanced interface provides access to the programmable pipeline on the new camera architecture [Lindsay and Agu 2009]. The development interface is a tool that provides the building blocks for assembling new camera pipelines. We discuss the process and design decisions that were used in implementing the camera interfaces. We also describe how user tasks were modeled and the types of interaction the interfaces provide. This paper also explains the implementation details of building the three interfaces, along with an Expert User study to validate our design.

Keywords: HCI, Computational Photography, Mobile Graphics

1 Introduction

Traditional cameras, such as Point & Shoot cameras, are tools for capturing scenes that are visually accurate. In a separate project from the one described in this paper, we developed a new camera architecture and prototype that provides full control over how the camera renders an image [Lindsay and Agu 2009]. Our new camera architecture provides a mechanism for camera users to write small programs, called shaders, which influences how the camera renders a photograph. As a result of the additional capabilities of this new camera architecture, traditional camera User Interfaces (UI) will no longer suffice. This is due to the complexity associated with providing programmable features for non-programmer camera users (users with little or no shader programming knowledge). Therefore, a new camera interface, as well as a development tool for non-programmers and programmers alike, had to be designed.

To facilitate the use of our camera's programmable features, we designed and developed three camera interfaces. The first interface was the development tool, which we call the Workbench, is used for creating new camera pipelines. The Workbench is a standalone application that allows the camera user to define the stages of the camera pipeline through a series of filters. The filters, which are implemented using Camera Shaders [Lindsay and Agu 2009], are small programs that are executed on the camera's graphics processing unit (GPU). Each filter takes as input an image and returns a modified version of that image to the next filter (see figure 1). The second interface, which runs on the camera, is the traditional camera interface. This interface allows the user to interact with the camera in the traditional sense, by allowing the user to take pictures, review and edit the pictures, and modify the camera settings. The third camera interface, which also runs on camera, is the advanced interface. This interface allows the camera user to choose, activate, and configure any of the camera pipelines created with the Workbench application. Once activated the new pipeline manipulates any captured image, and stores the results on the camera for review and editing within the traditional camera interface.

In this paper, we discuss the human-computer interaction decisions used to develop the interfaces, the programming paradigm for the pipelines, and the constraints imposed on the design due to the target platform. Furthermore, we discuss the implementation of the interfaces, including a survey of UI frameworks, programming languages, and software packages used to implement the functionality of the camera. Because this is a new type of interface, an expert user study was conducted in order to validate the design of the camera interfaces. Finally, we discuss the future work of this project based on the findings of the user study as well as other features that were not included in the original design of the camera interfaces.

2 Interface Design

2.1 Overview

The traditional design philosophy for shader development software follows a Directed Acyclic Graph (DAG) or other tree representation [Slusallek and Seidel 1995; Apodaca and Gritz 1999], which represents the inter-relationship of the shaders comprising an effect. For our shader development design, we chose to use a filter-based approach such as described in [Bennett and McMillan 2003], which and can be thought of as a decorator design pattern [Gamma et al. 1995][1]. It organizes filters in a sequence where the result of one filter is the input to the next. This filter design philosophy is consistent with the camera/photography community's philosophy, as filters (physical and effects) are generally used to modify images, such as those placed in front of a lens or used in Photoshop.

Our filter-based design is also consistent with the programming paradigm known as Stream Processing as described in [Owens et al. 2000], where a kernel (analogous to filter) takes as input a stream of data (image analog) and produces an output stream. In our case the streams are the input and output data from each filter. Thus each filter acts like a kernel, applying an operation on its entire input stream and producing an output stream.

2.2 Workbench

The primary development tool for creating and modifying camera pipelines is the Workbench application. The main goal of the Workbench is to allow the user to organize a series of filters and their properties to form a new camera pipeline. The Workbench interface provides a toolbox widget (figure 2, on the left hand side) to organize the filters available for modifying pipelines. The other widgets in the Workbench interface include the camera pipeline view, properties editor, and a preview window. When designing

*e-mail: clindsay@wpi.edu
†e-mail:gogo@wpi.edu
‡e-mail:emmanuel@cs.wpi.edu

[1]From this point forward we will refer to shaders as filters to avoid any confusion unless describing a feature specific to shaders

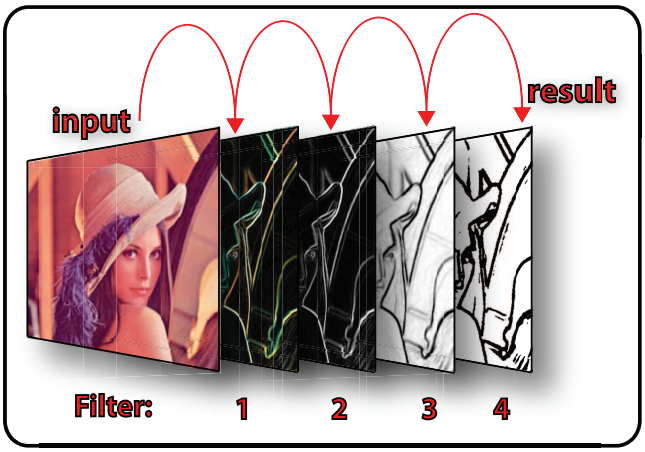

Figure 1: *Input image with a series of filters applied. Each filter is implemented using a Camera Shader.*

the Workbench application, the interface was centered on the filter philosophy previously mentioned. Each of the four components of the Workbench interface is purposed for organizing and editing filters, therefore they were arranged in a location that attempts to facilitate ease of use. For example, each component was placed in a location that promoted a left to right type design, where an operation begins at the left-side of the Workbench with the tool box and generally ends at the Properties Editor at the right-hand side.

To modify the pipeline, the user drags a filter icon onto the pipeline view in the desired location[2]. The location of the pipeline is directly adjacent to the toolbox, which allows dragging and dropping of filters with few mouse movements facilitating its ease of use. Additionally, the Toolbox is organized into a series of shelves[3], where similar filters can be organized into logical groups defined the user.

In the pipeline view, each filter is displayed with its name, state, and an icon representative of its type. To provide visual feedback to the user regarding the state of the filter, especially when a filter is not active, its checkbox will be unchecked, the filter icon will be grayed out, and the filter will not influence the resulting image in the pipeline. To get a more detailed view of the each filter within the pipeline, the user can select an individual filter by clicking on the filter icon in the pipeline which will load its properties in the properties editor.

The property editor allows the user to change the input values or parameters of the individual filters that are active within the pipeline. The filter properties are the input values for each filter and thus influence the behavior the algorithm used to implement the filter. When a filter's property is modified, its value is automatically updated in the pipeline and the result of the change appears immediately within the preview window. The preview window gives the user an indication of how the current state of the pipeline would look given the sequence of filters and their properties applied to a static image.

[2]The camera pipeline is a sequence of filters in which the order determines the sequence of operations. Therefore dropping a filter in a certain location can have varying effects depending on the order.

[3]The term shelf is a metaphor used by the visual framework described in section 4

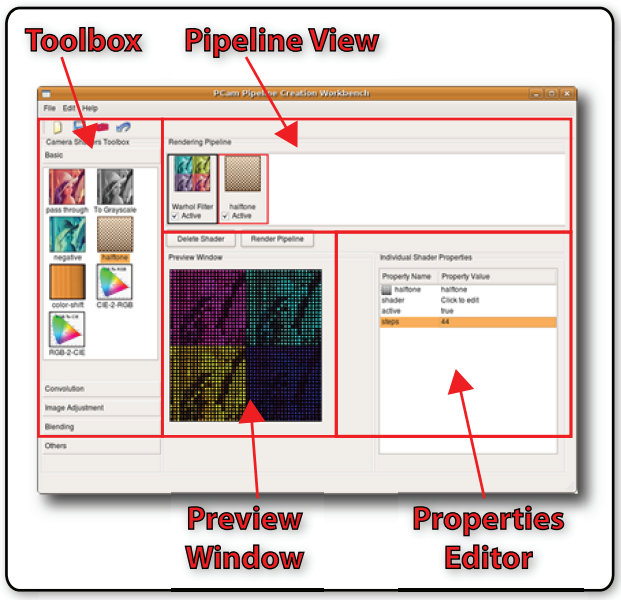

Figure 2: *The Workbench application provides an interface that allows users to create and modify camera pipelines. The interface is organized in a left-to-right fashion to facilitate ease of use and minimal number of steps to perform operations. The interface is comprised of four widgets, toolbox, pipeline view, properties editor, and preview window.*

2.3 Traditional Camera Interface

Unlike the Workbench interface, which is a stand-alone application, the camera has two distinct user interfaces, the traditional camera interface and the advanced user interface. Each of these interfaces was designed to be run simultaneously on the camera. The traditional camera interface provides the functionality for acquiring and editing photographs, similar to the features provided with commodity cameras. Like commodity cameras, our traditional camera interface is further divided into a collection of sub-interfaces called Picture Mode, Picture Review, and Camera Settings. Each of the sub-interfaces are displayed using tabbed windows, which are be brought into focus by clicking on the respective tab name (figure 3 provides a screen shot of these tabs).

The Picture Mode sub-interface (figure 3, left) controls the camera's photographic activities as well as providing an information display regarding the camera settings and the scene preview. The display of information within this interface is divided into two groups, the photography settings and the camera's state. The photography settings that are displayed include shutter speed, exposure, orientation, and color space. The second group displays information about the camera's state, such as the mode, the date/time, if the flash is active and a battery life indicator. Situated to the right of the settings is the preview window, which shows the current field of view for the camera. The second sub-interface is the Picture Review interface (figure 3, center), which provides the user with a view of pictures that have already been taken. This interface also provides some basic tools for editing the images such as adjusting the saturation, contrast, and brightness. Once the pictures have been edited, the user can save those edits by clicking the "Save" button or delete the image if the image is no longer desired.

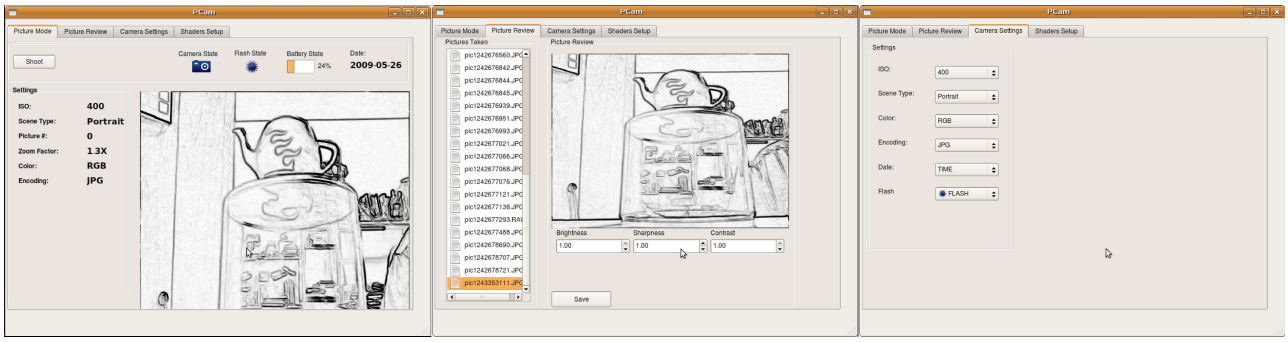

Figure 3: *The Traditional camera interfaces; left is picture mode, center is picture review, right is the camera settings.*

2.4 Advanced Camera Interface

The Advanced Interface provides additional functionality beyond that of the Traditional Interface, and allows users to define the camera's pipeline through the use of programmable filters. The filters are applied to images during the capture process and form a new camera pipeline in lieu of the traditional static camera pipeline found in today's cameras. As illustrated in figure 1, each filter applies its modification of the image then the results are fed into the next filter. The Advanced Interface provides four components to the user for configuring and previewing the available camera pipelines. These components are the Pipeline Selector (labeled "'Available Pipelines"' in figure 4A), the Pipeline View (figure 4B), the Properties Editor (figure 4C), and a preview and information display (figure 4D).

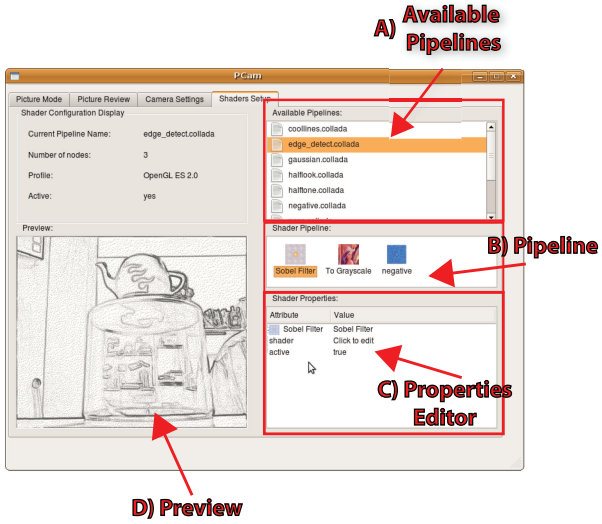

Figure 4: *The advanced user interface. Top right is the component where the use can select the available pipeline (exported from the Workbench). Below the pipeline selection, are the editable properties of the selected pipeline.*

As can be seen from figure 4, the UI components for selecting and configuring the pipelines are all located on the right-hand side of the interface. They are organized in a top-down or hierarchical fashion, where the top component is used for selecting the desired pipeline, the middle component is used for selecting the individual stages in the pipeline (determined by selecting a filter), and the lower component shows the properties of the selected filter. The user can activate a pipeline to be used on the camera by selecting the desired pipeline from the list of "Available Pipelines" (figure 4A). By default the first filter in the pipeline is selected and its properties are displayed in the bottom component (figure 4C). If the user desires to display and edit the properties of another filter within the current pipeline, the user can do so by selecting a different filter (figure 4B). When a property is edited, the value is automatically activated in the camera pipeline. This saves the user from performing any unnecessary steps in order to apply the change.

The Advanced Interface also contains a preview window and a component for displaying the current pipelines meta-data and rendering properties. The preview window displays the results of the current pipeline on on the camera scene. The preview window will update the image when a change to any filter is made, otherwise it remains static. In addition to the preview window, the properties relating to the pipeline as a whole are displayed above the preview window. The information displayed there tells the user information about the pipeline, specifically the number of filters used, the name of the pipeline, the rendering quality, and whether it is active.

3 Interaction Design

3.1 Task determination

For capturing an image using a programmable camera, there are two main tasks the user has to perform, first is creating a camera pipeline and second is using that pipeline to acquire images. The Workbench application was designed and implemented for creating new camera pipelines. Creating new pipelines can be further divided into three sub-tasks, pipeline building, pipeline fine-tuning, and pipeline storage. The second main task is acquiring images, which can be carried out via the following steps or sub-tasks, image sensing (activating the camera's capture capabilities), reviewing captured photographs, adjusting camera and pipeline settings, and changing the camera's pipeline. Each sub-task can be described as a series of atomic actions or small tasks accomplished in a single step, and which bring the user closer to completing the main task [Shneiderman 1986]. To aid in acquiring images, we designed a camera interface that can be mapped to the previously described sub-tasks. The rest of this section describes in more detail each of the main tasks and their corresponding sub-tasks.

3.1.1 Pipeline Creation Tasks (1^{st} Main Task)

The first high-level task of building a pipeline consists of organizing a sequence of filters, in a particular order, to achieve the desired rendering effect. The overall process is described in the flow diagram in figure 5. We can describe the atomic actions the user needs to perform as dragging and dropping a filter located in the toolbox (see figure 2) onto the pipeline. Thus, building of a camera pipeline

is an iterative as well a progressive process, consisting of one or more of these atomic actions with each iteration getting the user closer to the desired result. The user has completed the task when the output of the preview window displays the desired effect.

The second task is pipeline fine-tuning (property editing), which can be done simultaneously with the building of the pipeline and/or after the pipeline is assembled. The tuning task is iterative in nature as well, with the repeated performance of atomic actions until the desired effect is displayed in the preview. The atomic actions in this case are the editing of the properties and removing and reordering of individual filters. The properties of each filter can be edited within the properties editor (see figure 2) after selecting a filter. Reordering the pipeline requires the user to select and drag and drop an existing filter within the pipeline to another location within the pipeline, or remove it entirely by clicking the delete button underneath the pipeline.

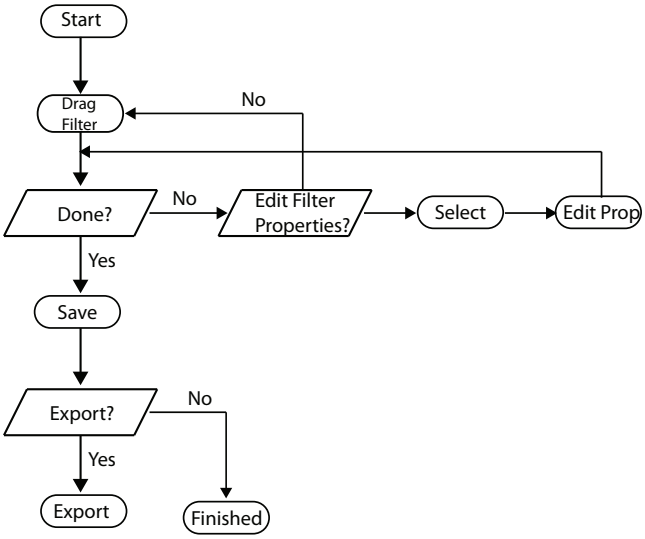

Figure 5: *Flow chart diagram depicting the tasks to create a new pipeline using the workbench application.*

The final task of storing or "'exporting"' the pipeline can be performed after the pipeline is assembled and its parameters tuned. Storage can be achieved in two ways. First, a local copy of the pipeline can be saved by clicking the "Save" icon. The other persistence method is exporting the pipeline to the target camera by clicking the "Export" icon. Saving the pipeline allows the user to save a pipeline locally for further editing at a later date without exporting to the camera. Exporting the newly created pipeline converts the pipeline to the camera's internal format and pushes it to the camera, making it available on the target camera. Modifying a pipeline that has been exported to the camera, is achieved by opening an existing locally saved pipeline and re-export it using the same pipeline name.

3.1.2 Image Acquisition Tasks (2^{nd} Main Task)

On the camera, the main task is capturing an image. This high-level task can be divided into three subtasks when using traditional cameras; image capture, reviewing captured photographs, and adjusting camera settings. For a programmable camera, we add a separate subtask, which is activating a new camera pipeline. The image capture task has requires the user to perform a physical action related to aiming the camera and clicking the capture button. The review task is slightly more complicated, where the user is required to navigate to the Picture Review sub-interface where they can review the picture. Reviewing the picture involves three separate tasks, selecting the picture, editing some basic image properties, such as contrast, sharpness, and brightness, and saving the modified image. The third task, involves editing the camera's capture property on the Camera Settings sub-interface. Selecting the desired property and modifying the value using the pulldown widget results in modification of the property. Once the property value has been changed within the widget, this new value is automatically applied and thus reflected elsewhere in the camera's UI and future captured images.

The fourth and final task is modifying and changing the camera's pipeline. After navigating to the Shader Setup sub-interface, the main subtask is to select and modify a camera pipeline for capturing images. This task itself can be divided into two separate subtasks as well, selecting the pipeline and editing the properties of the filters within the selected pipeline (see figure 4). Selecting a new pipeline requires an atomic action by the user, which is the user selecting the desired pipeline. This action automatically activates the new pipeline, thereby causing future images captured by the camera to be rendered using this new pipeline. The second task of modifying the pipeline requires two atomic actions, the first being selecting the filter with the properties that need editing, and the second being editing the property in the properties window (see figure 4). The set of tasks within the Shaders Setup sub-interface is similar to the Workbench with respect to editing the camera pipelines, due to using the same editing tasks and a similar layout.

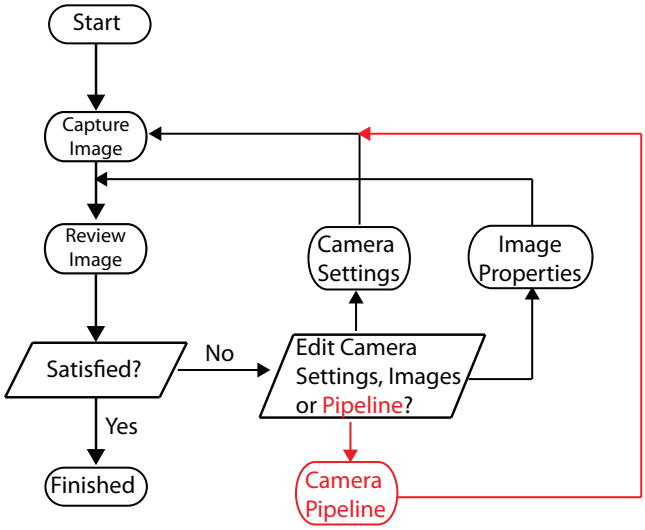

Figure 6: *Flow chart diagram depicting the tasks to acquire an image using the camera interfaces. The black text/lines denote traditional camera interface tasks and the red denotes tasks required for the programmable camera interface.*

3.2 Interaction Style

Most of the user interaction style employed by the Workbench and camera interfaces can be classified as either direct manipulation or menu-based manipulation as defined by [Shneiderman 1986]. The direct manipulation interactions are the actions the user needs to employ to visually create and manipulate camera pipelines. For example, dragging a filter from the Toolbox to the pipeline changes the pipeline, thus indicating that the user wants to include that filter within the pipeline. This direct manipulation interaction style allows the user to "physically" act as they are dropping filters to influence the pipeline. Direct manipulation is also employed with a

subset of the properties of the filters, such as using a color picker, value sliders, and spin boxes to edit properties.

In addition to the developed interfaces providing direct manipulation, the Workbench and camera interfaces provide additional feedback in several different ways. The main feedback mechanism used within the Workbench application and camera interfaces is the use of preview windows for previewing changes to the camera pipelines. The preview windows provide visual feedback in the form of a modified image that reflects the changes made by the user. For example, the user might assemble a new pipeline that contains a single filter for converting captured color images to monochrome images. Then by adding a Negative filter, the user will be able to preview the effect of monochrome images having the pixel values inverted. This immediate feedback could be a powerful means for rapid prototyping of pipelines because it helps to facilitate the often complex task of building a pipeline of filters by constantly providing feedback so the user can be guided.

3.3 Semi-automated Filter Creation

The Workbench and camera interface applications were intended to be designed with a shallow-learning curve with respect to shader writing (used to implement filters). We accomplished this through the use of visual manipulation metaphors by allowing the user to drag and drop filters, represented by icons, into the pipeline. The Workbench application provides a set of foundation filters provided within the Toolbox area of the Workbench application that can be used to create pipelines without the need to modify the shader code that implements the filters. The Workbench application provides enough filters that a majority of the needs of users for creating pipelines is satisfied, such as color conversion, image processing, image manipulation, and several artistic operations. This allows users with beginning to intermediate level shader writing experience to focus on creating the camera pipelines without forcing them to learn the underlying graphics shader technology. To complement the shallow-learning curve design, we also designed the Workbench application to provide advanced and expert users the flexibility to influence the underlying shader code and create new filters by providing two additional interfaces. The first is the shader editing capabilities provided in the properties editor component, and the second is the XML/GLSL configuration used by the Workbench. The shaders (filters) are implemented using the standard GLSL shader language to provide cross-platform compatibility. The shallow-learning curve with advanced editing capabilities strives for ”Universal Usability” while maintaining ”Internal Locus of Control” for expert users as described in the principles of [Shneiderman 1986].

3.4 Visual Appearance

The color palette of the interface was somewhat conservative, because we intended user attention to be focused on the rendering rather than being distracted by the color scheme. The use of a small number (3-4) of colors within the color palette of the application allows the color stimulation from the shader icons, preview windows, and other color indicators to appear more prominent. This draws the attention of the user to areas of importance when the user is interacting with the interfaces. This is particularly important when utlizing the Workbench application to design camera pipelines that requires the user’s to focus attention on the color of the rendering. One place where color was used to draw attention was in the selection of filters within the pipeline. When the user selects a filter, changing the border color and background to red highlights the target filter. This allows the user to discriminate between the different shaders within the pipeline even though they may contain complex colors themselves as described in [Shneiderman 1986].

4 Implementation

4.1 Overview

The implementation of both the Workbench application and the camera interfaces went through two phases before completion, the review phase and implementation phase. The review phase consisted of evaluating three separate UI frameworks for possible implementation of the Workbench and camera interfaces. The second phase, the implementation phase, consisted of implementing the Workbench and camera interfaces using our choice of UI framework. For this project, we chose the Qt framework [Nokia 2009] due to availability on the target computing platform (Beagleboard [Beagleboard 2009]), the feature set it provided, and its compatibility with the chosen programming language (Python) memory footprint. In the following sections we describe the review process, the implementation, and the software and hardware used to implement the Workbench and camera interfaces.

4.2 UI Server Comparison

Each choice of UI framework depends on an underlying graphics rendering environment in order to visually display their widgets. Because we are using Linux as the target operating system kernel (Angstrom as the OS), we have three choices for graphics rendering environments, the X windows system, Qtopia (Trolltech/Nokia), and using the frame buffer directly known as DirectFB. By far the most widely used environment on all Linux and Unix kernels is the X windowing system which provides a client/server configuration for graphics rendering. Qtopia is a self-contained windowing system implementing the Qt UI framework and is targeted for embedded systems. Several UI frameworks have the capability to directly access the frame buffer, such as GTK, thereby bypassing the overhead of an intermediate hardware abstraction layer. Because each framework considered depends on an underlying graphics environment, these dependencies have to be taken into consideration when choosing. Table 1 provides a comparison of the criteria used for determining the best UI framework for this project.

X allows reuse of many existing UI framework for embedded UI thereby making the available number of frameworks much greater than other graphics environments. Some examples of UI framework currently running on the Beagleboard with X are Qt (not Qtopia), GTK, FLTK, Motif, and Java’s AWT. Because X was designed in a client/server paradigm, the network-oriented overhead associated with communication and multiple processes of the server and client is a concern. X servers are particularly useful in environments, which is require several graphical applications simultaneously, such as mobile internet devices (MID).

DirectFB provide a high-level abstraction of the Linux frame buffer interface. This option provides access to the underlying hardware via an abstraction but does not provide any graphics or rendering routines. This is a popular choice for developing new UI frameworks, such as with Clutter [Project 2009] for leveraging OpenGL [SGI 2009] for rendering UI widgets. It requires drivers for OpenGL ES 2.0, which are currently unavailable in the public domain for the Beagleboard. This can be very fast as in future releases may take full advantage of graphics hardware for acceleration.

The embedded version of the QT UI framework called Qtopia [Nokia 2009] is capable of running directly on the frame buffer, enabling ported or new applications developed using Qt to run on an embedded system with incurring the overhead cost of using X. Qtopia is an integrated graphic software stack providing the graphic environment, support libraries, and widget toolkit. Qtopia pro-

vides an optimized version (somewhat restricted) version of the Qt framework to run on embedded platforms with a memory footprint around 5MB. Qtopia and Qt on X provide two options for running Qt based applications on embedded systems. Future versions will also provide graphics acceleration via graphics hardware, thereby allowing for faster rendering and integrated graphics widgets.

4.3 UI Frameworks Review

Before determining the UI framework that would be used to implement the Workbench and camera interface, we reviewed three viable options for the implementation. These frameworks were Qt, Flex [Adobe 2009], and Glade [Glade 2009] with GTK [Gnome 2009]. Qt is a cross platform UI framework originally developed by Trolltech (now owned by Nokia) in C++ with Python bindings. Flex is an open source framework based on Adobe's Flash for developing rich Internet applications to be run within a browser that is and uses an XML variant called MXML. Glade (User Interface Designer) and GTK+ are two complimentary tools for creating cross platform UIs written in C++ but has many bindings. Initially, we started the development of the interfaces using all three frameworks.

Each of the available implementation frameworks we considered were viable options. Therefore, we used a method of elimination to determine the best framework for this project by considering performance, available multi-media widgets, and if it was currently working on the Beagleboard ditribution operating system (Angstrom). We conducted an informal review to determine which framework would be the least viable option for our implementation. Considering that Qt provided a mature framework supported by a large company made it an attractive choice. But one very important criterion we had to consider was how it would perform in an embedded environment. Qt is ideal for embedded environments because it offers a specific version of its runtime for embedded systems called Qt Embedded. The other two choices did not offer an embedded option, although Adobe Flash (proprietary version of Flex) has been implemented on several mobile devices but currently not the Beagleboard. Due to the lack of Flex/Flash plugins for the Beagleboard browsers, it was currently impossible to run a Flex interface on the embedded system itself and required a web browser on an external machine in order to view the camera interface, which invalidated this choice.

The programming model for Flex was considerably different than the other two (considering the web based client/server model) choices. GTK and Qt could be developed using a single program with separate classes which is more traditional for desktop and embedded programming. Flex required a webserver that was configured to run CGI scripts (for the Beagleboard, this was Apache). In addition to providing a familiar desktop programming model, QT also provided specific classes implemented to provide a Model/View/Controller programming implementation.

Another determining factor was that the filters (shaders) implemented for this project relied heavily on the use of OpenGL for rendering. This means that the UI framework would have to have an OpenGL compatible display widget. QT and GTK had such a component, but at the time of implementation it was not clear that Flex did. Because Qt had embedded support, was mature, provided additional programming features (MVC), and was OpenGL compatible, it was chosen over the others.

5 Qt UI Implementation

As previously mentioned, for implementing the UI of the Workbench application and camera interface, we decided to use the Qt UI framework. From Qt version 4.2 and above, the framework provides groupings of classes specifically designed to be used within a Model/View/Controller programming paradigm. Both interfaces used this paradigm for organizing the implementation software (figure 7). The View consisted of the QT UI widgets. The Controller was Python code designed to react to and instigate actions that constituted the application logic. A light-weight database was used as the Model. This separation of components has the advantage of decoupling the UI from the code, and provides a clear segregation of the application components, thus minimizing interdependency and facilitating testing, refactoring, and reuse.

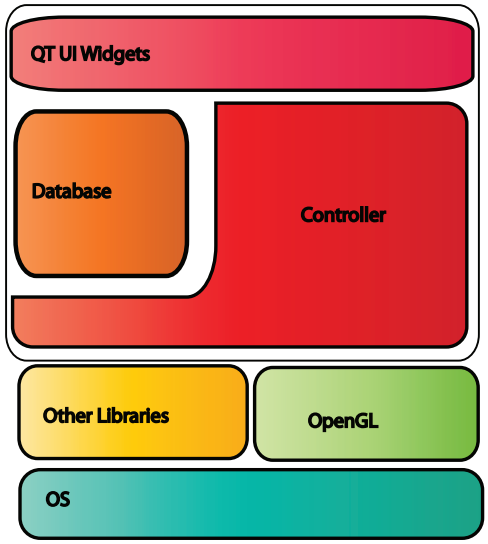

Figure 8: *The software organization of used for both the Workbench applications the camera interfaces. The top layer represents the UI components used to interact with the user, the controller provides the functionality of both interfaces and interaction with the database. Direct acess to the Opengl rendering framework was used to implement the shaders.*

The Model of the application consisted of the shaders and their associated properties, which we used as filters. These properties were loaded and stored within a lightweight database provided by Qt called the QtStandardItemModel. Each shader was loaded and converted into a subclassed QtStandardItem called PcamStandardItem. Each item provided a tree structure to store the property values of the corresponding shader. A root item was created, and each node below the root was a shader property. Once the tree was completed, it could then be inserted into the database. The QtStandardItemModel provided many convenience functions for querying, listing, and updating various properties of each item.

The View of the application was implemented with two kinds of widgets, "'simple"' and "'complex"' widgets. The simple widgets, such as labels and buttons provide a basic look and feel with minimal interaction capabilities. The complex widgets provided a way of viewing models created with the QtStandardItemModel database. The complex widgets constituted the View portion of the MVC paradigm used because they allowed the view of the model to be separate from the actual storage of the items. The view widgets used were the Qt provided QtTableView and QtListView. In addition to providing views, these widgets allowed a certain level of interaction with the model through selection and drag and drop operations. The views also used delegate classes to provide special functionality such as non-standard rendering and interaction. These delegates, called QtStyledItemDelegates were used to provide customized rendering of the pipeline (QtListView), which include an

	Api/Server	Memory	Multimedia Widgets	Working on BB	Scripting	UI Designer	RT
1	**Qt on Qtopia**	**5MB (No X)**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
2	Flex (Flash)	<1MB (No X)	Yes	Yes	Yes	No	Yes
3	Gtk	12-15MB	No	Yes	Yes	Yes	Yes
4	FB (No X)	N/A	N/A	N/A	N/A	Yes	N/A

Table 1: *A comparison of various features of four different UI frameworks we considered for implementing the camera interface.*

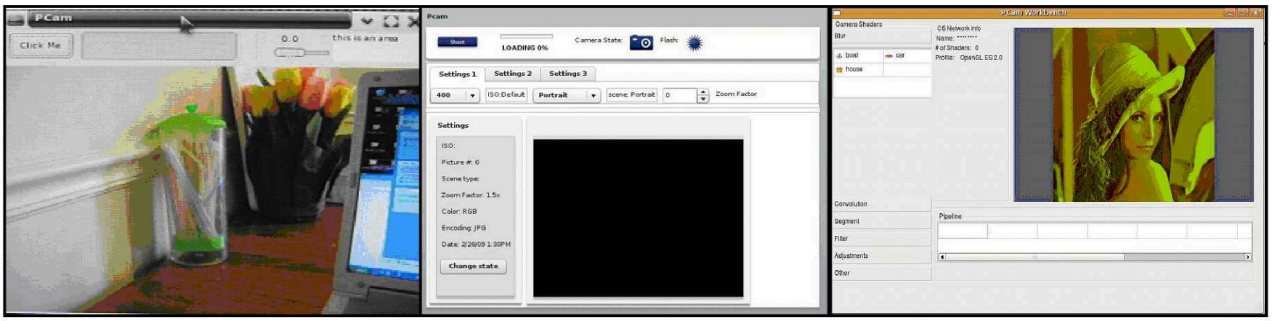

Figure 7: *Three screen shots of early UI from the each of the three sampled frameworks. Left, is the Glade/GTK+ running on the Beagleboard. Center is the Adobe Flex interface running within a browser. Right is the Qt interface.*

image, name, and activation checkbox for each filter. The properties of each camera shader within the pipeline were displayed using the same model but a different view (QtTableView). A customized QtTableView was implemented to provide a hierarchal view and editing capabilities of the shader's properties. The View was a vertical table with the first column being the name of the property and the second column being the value of the property. Each property had a specific datatype (color, float, integer, enum, image, etc.) that required a specialized editor to be implemented. Another complex widget used within the interfaces was the OpenGL rendering widget. The OpenGL widget converted the current pipeline into a visual rendering using the underlying graphics hardware (GPU).

The controller part of the interfaces facilitates the interaction between the UI widgets, the databases, and other libraries. The controller takes interaction commands from the UI and performs a specific action related to editing this property, such as providing a widget to the user for editing. These actions could be the movement of shaders from the toolbox to the pipeline, editing a shader's property, or direct manipulation of the pipeline itself. The controller was implemented primarily in Python and facilitating interaction between Qt, OpenGL, and other libraries such as OpenCV (access/control of camera).

6 Programming languages & Standards

Independent of the UI framework, we used Python as the foundational programming language. Python is a high-level programming language used for general-purpose programming. It has a minimalistic syntax, a comprehensive set of libraries, and it is cross platform compatible. Because the intention is to utilize this project for the long-term interface for our programmable camera, cross-platform compatibility and bindings for each of the possible UI frameworks was an essential trait for our programming language choice. Another potential language choice was C/C++ but because of the difficulties in cross compiling and library availability on the Beagleboard, Python was the better choice of programming languages.

For representing shader assets we used the popular open digital asset standard called Collada. Collada is a file format standard used to exchange digital assets for rendering and image synthesis. For example, Collada is used to store 3D models and textures for games independent of the game engine or Digital Content Creation (DCC) tool used to manipulate them. Collada is based on XML and contains a wide variety of supported formats beyond just code for shaders, such as formatting of 3D models, textures, and scene graphs.

In addition to Collada, we used a simple XML based tree for organizing the Toolbox within the Workbench application. The XML format organized the default shaders into categories and supported meta-information as well as parameters for each shader. The Toolbox categorization and ordering is determined by an XML file, which is loaded at run time. When a particular shader within the pipeline is selected, a properties editor is activated. The properties editor allows the user to edit the specific properties defined by the shader as well as activating and naming the particular shader instance. If the user changes the parameter of a specific shader and decides to provide this as a new default shader, the user can drag the modified shader to the Toolbox and save it. This modifies the original XML file used to configure and load the Toolbox at the start of the application.

7 Software Packages & Other Frameworks

The implementation of the workbench and the camera interface relied on three other software frameworks for performing rendering and image manipulation Opengl, GLSL, and Python Image Library (PIL). OpenGL and GLSL (OpenGL shading language) provided the framework for implementing the camera shaders (i.e., filters) and PIL was used to perform image enhancements. The Workbench application and camera interface use both for providing previews of the current pipeline. In addition to providing a preview, the camera interface uses these two frameworks to render the final image.

After the captured image has been rendered, the images can be further modified within the camera interface in the "Picture Review" interface (see figure 3). There the user can adjust the brightness, sharpness, and contrast of the images that have been captured to further enhance the image. The implementation of the basic enhancements was done using the PIL (Python Image Library) module for python. The PIL module also provides basic conversion capabilities

for images going between standard formats, such as JPG, GIF, and PNG, and the image formats used within Qt (Qimage, Qpixmap).

8 Evaluation

8.1 Expert User Study Design

To validate the design of the Workbench and camera interface that were implemented, we developed a user study. The IRB-approved user study was an Expert User study, where an expert camera user was recruited to use and evaluate the interfaces of the camera and Workbench application and provide constructive feedback regarding the layout and operations needed to perform camera related tasks. The expert's feedback provided a way for us to validate the design of the interfaces as a whole. The primary reason for validating the UI was to verify that we were able to provide additional camera functionality while preserving the user interface design within a typical camera. To standardize the study, we developed a protocol, which outlined the steps the expert user should take when evaluating the interfaces. This allowed for focused feedback from the user with respect to the layout, organization of tasks, and preservation of expected camera functionality. The protocol outlined three areas of focus for the evaluation, the normal camera interface, the advanced camera interface, and the Workbench. The evaluation was done on a simulated camera system using a laptop with an embedded web camera. The study was recorded via video camera for review.

To evaluate the typical camera operations, the user was instructed to perform tasks related to normal camera functions, such as taking a picture, reviewing and adjusting captured images, and modifying the camera settings. The interface study allowed the user to take as many pictures as the user wanted while transitioning from one interface to another as the user would on a commodity camera. When the user completed taking pictures, several questions related to the interface of the camera were asked. When evaluating the typical camera interface, we provided the user with the traditional camera pipeline only.

The next task performed in the user study was the evaluation of the advanced camera interface, which is used to select and configure alternate camera pipelines. The user was given access to the advanced interface, and was instructed on how to select and configure new camera pipelines. Then the user performed the same operations as with the normal camera interface, but with the new pipeline. The user was allowed to take pictures, adjust the images, change and reconfigure the pipeline at any time during the study. Again, when the user completed taking pictures, several questions related to the interface of the camera were asked.

The third part of the user study involved the user performing pipeline-creation tasks on the Workbench application, such as building, previewing, and exporting a new camera pipeline. The user was asked to create a new pipeline using the existing filters available within the Toolbox area of the Workbench application. When the user was sufficiently satisfied with the pipeline that was created the user was asked to export the pipeline to the camera. When the user completed evaluating the workbench application, several questions related to the Workbench interface were asked. To conclude the study we asked several questions with respect to the interfaces as a whole.

8.2 Results of Expert User Study

The Expert User study provided informative feedback with respect to the design of all the interfaces including the Workbench application. The organization and layout of the normal camera UI (i.e. non-advanced UI) was indicated to be consistent with normal camera operations seen in commodity cameras. The most prominent issue indicated by the Expert User was the lack of feedback with respect to taking pictures. When the user performed the actions for capturing images, there was no auditory or visual feedback when a picture was actually taken. For example, when taking a picture with film-based cameras, the mechanics of the shutter provided an auditory "click" and subtle vibrations, Digital cameras mimic this cue by providing a synthetic auditory "clicking" sound and sometimes blacking out the preview screen momentarily to simulate a shutter.

For the advanced operations, such as selecting and tuning a new camera pipeline, the user indicated that the additional functionality and layout did not over complicate the design and operation of the traditional camera functionality. But two issues were raised with respect to the interaction with the advanced functionality of the camera: 1) preserving the original image (traditional pipeline) was desirable, and 2) making the normal interface aware of the advanced features. For the first issue, the user indicated that they would prefer to view the new pipeline in real-time as well as saved the rendered image while also retaining an un modified or raw image as well. The second issue stemmed from the use of the normal interface, specifically changing the scene type rapidly. The user indicated that it would be more natural for new pipelines to be available for selection within the Camera Settings sub-interface under the "scene types" setting. This would integrate new pipelines into the modality of new scenes, which the user considered new pipelines to be. This would also alleviate the need for additional steps for switching between different scenes and different pipelines. The user suggested allowing the advanced interface to selectively save new pipelines to the list of pre-existing scene types.

Most of the constructive feedback and observations came from the Workbench application. The Expert User was familiar with cameras, video editing, and various digital content creation tools. Although the Expert User was aware that shaders could be written, the expert had no experience in doing so. Within several seconds the Expert user was able to assemble a new pipeline in the Workbench application and when asked the layout and design seemed to be easy to use and produced results rapidly. One indication of a flaw in the layout was the fact that the user repeatedly tried to drag and drop shaders onto the preview window despite being instructed that the filters needed to be dropped on to the pipeline. When asked, it was indicated that the preview window in the Workbench application seemed like the logical place to drop the shaders, probably due to the visual layout and Direct Manipulation style of the application. The user also indicated the need for reversing or "undoing" actions as was typical in other creation tools. Undo capability was indicated a missing feature that was highly desirable. The final constructive criticism from the user was that it would be useful for the application to indicate the progressive changes each filter made by providing a thumbnail image for each filter indicating how the pipeline looked at that point in the pipeline.

9 Future Work

9.1 Improvements From Evaluations

The user study provided an evaluation of the Workbench and camera interface software that produced valuable recommendations for improvements. While some of the recommendations will be easily implemented, a number of them will require more involved refactoring of the code base. Below are some improvements for future work:

1. Workbench: Reversal of Actions (undo)

2. Workbench: Drag to Preview window
3. Workbench: Each pipeline stage has thumbnail preview
4. Camera: Integrating New Pipelines into ”scene type” pull-down for ease/speed
5. Camera: Needs indication of a shot (sound/shutter open/close animation, etc)
6. Camera: Retention of original and modified images

9.2 Camera Control Scripting System

Control scripting for programmable cameras allows users to control the action of the camera by providing a script that triggers camera functionality in a predetermined way. The scripting system is a set of camera commands and logic that allow the user to pre-program the steps the camera will take when a shot is taken. For example, if the user wants to take an High Dynamic Range photograph by defining the exposure of the camera then the user would write a script that would take a sequence of photographs with different exposure settings, run a particular camera pipeline on the photographs, and store the results.

The scripting system can be used to control the features of the camera such as the shooting action, flash, as well timing. It can also be used to control the camera's photogenic properties such as exposure, shutter speed, and focus. Such systems already exist in various forms, most predominately in the Canon Hackers Development Kit (CHDK) [CHDK 2009] as an unauthorized scripting system for Canon cameras. For some time now, camera manufacturers have made digital still cameras (DSC) with scripting capabilities such as the Kodak DCS260 (with Flashpoint's Digita OS), the Minolta Dimage 1500 and the more recent Canon cameras with Digic II and III processors. Scripting within cameras gives the camera user the ability to trigger existing camera functionality by automating a sequence of steps, normally performed manually by the camera user, through the use of small programs called scripts. For example, the CHDK firmware ”enhancement” lets Cannon Powershot users write camera scripts that mostly emulate button clicks or menu selections the user would perform while using the camera.

9.3 Export System Improvements

Currently our system for exporting camera pipelines to the camera requires a two-step manual process. First, the user clicks the save button and a file dialog appears. The user has to find a location to store the pipeline on their file system. Once the pipeline is saved, the user then has to manually copy the pipeline file to the location of the camera system, which maybe on another computer. Instead we would like to automate the process of exporting that preclude the user from having to use the file system to store the pipeline file and instead copy it directly to an available camera.

In addition to simplifying the interface, we would also like to provide better adherence to the Collada standard for our pipeline exports. Ideally each pipeline would consist of a single Collada file, which embeds, the Camera Shaders, all the assets associated with each Camera Shader (e.g., textures), and any pipeline properties such as rendering quality.Also, each pipeline could include in the Collada file the scripting code associated with the pipeline.

9.4 OpenCL

OpenCL is a framework for executing stream-oriented programs that will be executed across multiple processors (GPUs & CPUs), specifically aimed at multicore processor platforms with mixed CPUs and GPUs. OpenCL will be an open standard such as OpenGL and OpenAL, therefore it will be supported on various platforms with CPUs and GPUs like mobile and embedded devices. Future versions may include support for OpenCL processing code instead of low-level GPU programs. In addition, the possibility of using OpenCL programs in conjunction with OpenGL programs exists. In order to provide maximum flexibility in terms of heterogeneous computations (general purpose and graphics) future implementations may contain OpenCL code to leverage both the CPU and GPU for graphics and general-purpose computations.

Appendix

9.5 Terminology

- Pcam - The design of a new type of programmable camera that allows for arbitrary and programmatic enhancement of static camera pipelines. It is intended as a replacement for traditional camera pipelines.
- *Interface* - Is a software application that is used to control a complex system of software and hardware that contains text or visual elements indicating state of the system and ways of influence/control that state.
- *Sub-interface* - Is a part of the interface that is separate from the other parts.
- *Framework* - A system of software used to provide a foundation for building applications.
- *Workbench* - Is the application designed and developed by use for the purpose of creating new camera pipelines.
- *Camera interface* - Is the application designed and developed by us to be used to control the camera.
- *Toolbox* - An area of the Workbench application used for storing the default shader/filters for building new pipelines.
- *Component* - A dedicated area within the application. This area is generally used to perform a particular function or task, such as a preview window or editor.
- *Shaders* - A set of instructions, usually in the form of a small computer program or function that describes how part of an image is going to be manipulated.
- *Camera Shaders* - A shader that is processed on-camera in order to manipulate the image captured by the camera.
- *Filters* - A process that is designed to enhance or manipulate an image in someway. Also used a synonym for shaders.
- *User Interface (UI)* - Visual software that is used to interact with the software's user.
- *Programmable Cameras* - A new type of camera that replaces the static pipeline of today's cameras and replaces it with a programmable pipeline.
- *Fragment Shader* - A small program that is executed on the GPU that manipulates fragments and converts them into potential pixels.
- *Vertex Shader* - A small program that is executed on a GPU that manipulates the input vertices of a 3D representation of a scene.
- *Image Processing* - The process of applying an operation on an image for the purpose of changing the image in a meaningful or artistic way.

- *Camera Pipeline* - Series of stages that a captured image goes through to render the final image. These steps generally help to convert the image from a low-level format to a representation that is amenable to interpretation by humans.
- *Model/View/Controller Paradigm (MVC)* - MVC is a software engineering programming paradigm that separates software components according to their role in the application. The three roles are the Model (data and associated functionality), Controller (application or business logic), View (the UI component).

9.6 Questionaire & User Study Script

Below is a copy of the script used for the expert user study (4 pages):

User Study Script:

This study looks at the usability of an advanced camera interface. This study will walk you through several steps with the goal of applying image processing effects to images on a new type of programmable camera. The steps below should be followed in order.

Step 1 – Generate a pipeline that does simple edge detection. You will use three shaders organized in a pipeline: Grayscale, Sobel Filter, and Negative.

Open the Pcam Workbench application. To generate a pipeline, you need to **drag and drop** shaders from the *Shader Toolbox* area to the *Rendering Pipeline* area (see figure 1). A preview of the how the pipeline will affect the rendering is visible in the Preview Window, which appears below the Rendering Pipeline area.

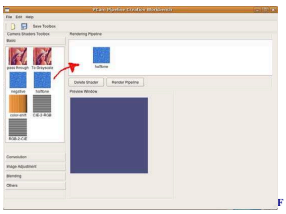

Figure 1: Workbench Application. To apply image filter to the preview window, drag and drop a specific filter from the Camera Shaders Toolbox to the Rendering Pipeline are.

Step 2 – Export the new pipeline to the Camera Interface.

Once you are satisfied with the look of the rendering for your new pipeline, you need to export the pipeline to the camera so the camera can use it to affect images captured by the camera. To do this, click on the **Save** icon in the application. This will bring up a file dialog for which you will have to name your pipeline. Type in a name for your new pipeline and click "**Save**".

Step 3 – Apply the new pipeline on the camera.

Open the camera interface application and select the *Shaders Tab*. This will open the part of the camera interface where you can load and configure the pipeline you saved in the Pcam Workbench application. Click on your pipeline in the *Available Pipelines* area. This will select and load the pipeline, and the camera will begin rendering this pipeline's effects. Once you've selected your pipeline, you can then

1

modify any parameters for that pipeline in the Shader Properties area.

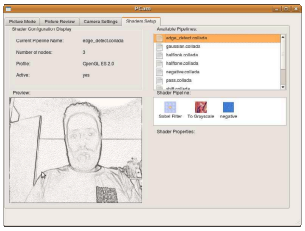

Figure 2: The camera interface's Shader Setup Tab. To activate a Pipeline created in the Workbench application just select the pipeline's name with the mouse.

Step 4 – Taking a picture with the new pipeline applied.

Once the new pipeline has been selected and configured, you can use it to take pictures. Click on the "**Picture Mode**" tab. This will change the interface to display a real-time image of what the camera sees with the new pipeline applied. You can now take pictures, which will be saved on the camera. The next step allows you to edit the image. Below is an image with of the interface with a real-time effect applied.

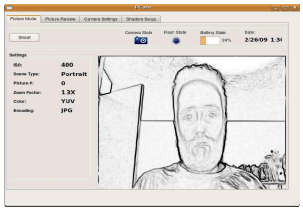

Figure 3: In the camera interface using the Picture Mode tab, you can take pictures by clicking on the button labeled "Shoot". A

2

References

ADOBE, 2009. Adobe flex 3. http://www.adobe.com/products/flex/.

APODACA, A. A., AND GRITZ, L. 1999. *Advanced RenderMan: Creating CGI for Motion Picture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

BEAGLEBOARD, 2009. The beagleboard is an ultra-low cost, high performance, low power omap3 based platform designed by beagleboard.org community members. http://beagleboard.org/.

BENNETT, E. P., AND MCMILLAN, L. 2003. Proscenium: a framework for spatio-temporal video editing. In *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, ACM, New York, NY, USA, 177–184.

CHDK, 2009. Canon hackers development kit. http://chdk.wikia.com/.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. No. ISBN 0-201-63361-2. Addison-Wesley.

GLADE, 2009. Glade - a user interface designer. http://glade.gnome.org/.

GNOME, 2009. Gtk+ is a highly usable, feature rich toolkit for creating graphical user interfaces which boasts cross platform compatibility and an easy to use api. http://www.gtk.org/.

LINDSAY, C., AND AGU, E. 2009. P-cam: A programmable camera back-end. Tech. Rep. 999999, Worcester Polytechnic Institute, September.

NOKIA, 2009. Qt a cross-platform application and ui framework. http://qt.nokia.com/.

OWENS, J. D., DALLY, W. J., KAPASI, U. J., RIXNER, S., MATTSON, P., AND MOWERY, B. 2000. Polygon rendering on a stream architecture. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, 23–32.

PROJECT, O., 2009. Clutter. http://clutter-project.org/.

SGI, 2009. Open graphic language. http://www.opengl.org/.

SHNEIDERMAN, B. 1986. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

preview of the picture you are about to take with the shaders applied is presented in the window. Here a negative shader is in effect.

Step 5 – Editing and saving photographs.

Photographs taken with the camera can be edited and saved using an interface similar to that of a traditional camera. To use this interface, click on the "**Picture Review**" tab at the top of the camera interface. Edit the previously taken photographs by modifying the brightness, saturation, and contrast using the provided Spinboxes.

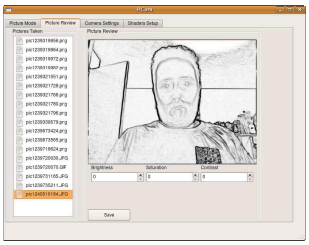

Figure 4: The camera interface's picture review tab. Once the picture has been taken, you can review the picture, adjust some of the basic properties, and save any changes.

Step 6 – Changing the camera settings.

*To change the camera settings, click on the "**Camera Settings**" tab to select the camera setting window. To edit the base camera settings such as shutter speed, focus, and encoding, select the appropriate settings from the pull-down menus.*

3

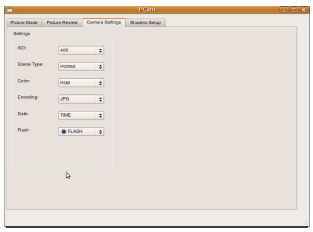

Figure 5: The Camera interface's Settings Tab. In this tab you can modify the camera settings.

4

SLUSALLEK, P., AND SEIDEL, H.-P. 1995. Vision - an architecture for global illumination calculations. *IEEE Transactions on Visualization and Computer Graphics 1*, 1, 77–96.